6.4 Cache Memory (Fully Associative Cache)

- Instead of placing memory blocks in specific cache locations based on memory address, we could allow a block to go anywhere in cache.
- In this way, cache would have to fill up before any blocks are evicted.
- This is how fully associative cache works.
- A memory address is partitioned into only two fields: the tag and the word.

6.4 Cache Memory

- You will recall that direct mapped cache evicts a block whenever another memory reference needs that block.
- With fully associative cache, we have no such mapping, thus we must devise an algorithm to determine which block to evict from the cache.
- The block that is evicted is the victim block.
- There are a number of ways to pick a victim, we will discuss them shortly.

6.4 Cache Memory (Fully Associative Cache)

- Suppose, as before, we have 14-bit memory addresses and a cache with 16 blocks, each block of size 8. The field format of a memory reference is:

```
11 bits  3 bits
Tag      Word
```

- When the cache is searched, all tags are searched in parallel to retrieve the data quickly.
- This requires special, costly hardware.

6.4 Cache Memory (Set Associative Cache)

- Set associative cache combines the ideas of direct mapped cache and fully associative cache.
- An N-way set associative cache mapping is like direct mapped cache in that a memory reference maps to a particular location in cache.
- Unlike direct mapped cache, a memory reference maps to a set of several cache blocks, similar to the way in which fully associative cache works.
- Instead of mapping anywhere in the entire cache, a memory reference can map only to the subset of cache slots.
6.4 Cache Memory (Set Associative Cache)

- The number of cache blocks per set in set associative cache varies according to overall system design.
- For example, a 2-way set associative cache can be conceptualized as shown in the schematic below.
- Each set contains two different memory blocks.

<table>
<thead>
<tr>
<th>Set</th>
<th>Tag</th>
<th>Block 0 of set</th>
<th>Valid</th>
<th>Tag</th>
<th>Block 1 of set</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00000000</td>
<td>Words A, B, C, ...</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>11111111</td>
<td>Words D, E, F, ...</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>01111111</td>
<td>P, Q, R, ...</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.4 Cache Memory

- In set associative cache mapping, a memory reference is divided into three fields: tag, set, and word, as shown below.
- As with direct-mapped cache, the word field chooses the word within the cache block, and the tag field uniquely identifies the memory address.
- The set field determines the set to which the memory block maps.

Implementation of Set Associative Cache

Performance among Caches
6.4 Cache Memory

With fully associative and set associative cache, a replacement policy is invoked when it becomes necessary to evict a block from cache.

An optimal replacement policy would be able to look into the future to see which blocks won’t be needed for the longest period of time.

Although it is impossible to implement an optimal replacement algorithm, it is instructive to use it as a benchmark for assessing the efficiency of any other scheme we come up with.

An optimal replacement policy would be able to look into the future to see which blocks won’t be needed for the longest period of time.

Although it is impossible to implement an optimal replacement algorithm, it is instructive to use it as a benchmark for assessing the efficiency of any other scheme we come up with.

The replacement policy that we choose depends upon the locality that we are trying to optimize--usually, we are interested in temporal locality.

A least recently used (LRU) algorithm keeps track of the last time that a block was assessed and evicts the block that has been unused for the longest period of time.

The disadvantage of this approach is its complexity: LRU has to maintain an access history for each block, which ultimately slows down the cache.

First-in, first-out (FIFO) is a popular cache replacement policy.

In FIFO, the block that has been in the cache the longest, regardless of when it was last used.

A random replacement policy does what its name implies: It picks a block at random and replaces it with a new block.

Random replacement can certainly evict a block that will be needed often or needed soon, but it never thrashes.

The performance of hierarchical memory is measured by its effective access time (EAT).

EAT is a weighted average that takes into account the hit ratio and relative access times of successive levels of memory.

The EAT for a two-level memory is given by:

\[ EAT = H \times \text{Access}_C + (1-H) \times \text{Access}_M. \]

where \( H \) is the cache hit rate and \( \text{Access}_C \) and \( \text{Access}_M \) are the access times for cache and main memory, respectively.

For example, consider a system with a main memory access time of 200ns supported by a cache having a 10ns access time and a hit rate of 99%.

The EAT is:

\[ 0.99(10\text{ns}) + 0.01(200\text{ns}) = 9.9\text{ns} + 2\text{ns} = 11\text{ns}. \]

This equation for determining the effective access time can be extended to any number of memory levels, as we will see in later sections.

Write through updates cache and main memory simultaneously on every write.
6.4 Cache Memory

- Write back (also called copyback) updates memory only when the block is selected for replacement.
- The disadvantage of write through is that memory must be updated with each cache write, which slows down the access time on updates. This slowdown is usually negligible, because the majority of accesses tend to be reads, not writes.
- The advantage of write back is that memory traffic is minimized, but its disadvantage is that memory does not always agree with the value in cache, causing problems in systems with many concurrent users.

- The cache we have been discussing is called a unified or integrated cache where both instructions and data are cached.
- Many modern systems employ separate caches for data and instructions.
  - This is called a Harvard cache.
  - The separation of data from instructions provides better locality, at the cost of greater complexity.
  - Simply making the cache larger provides about the same performance improvement without the complexity.

- Cache performance can also be improved by adding a small associative cache to hold blocks that have been evicted recently.
  - This is called a victim cache.
- A trace cache is a variant of an instruction cache that holds decoded instructions for program branches, giving the illusion that noncontiguous instructions are really contiguous.

- Most of today’s small systems employ multilevel cache hierarchies.
- The levels of cache form their own small memory hierarchy.
  - Level 1 cache (8KB to 64KB) is situated on the processor itself.
    - Access time is typically about 4ns.
  - Level 2 cache (64KB to 2MB) may be on the motherboard, or on an expansion card.
    - Access time is usually around 15 - 20ns.

- In systems that employ three levels of cache, the Level 2 cache is placed on the same die as the CPU (reducing access time to about 10ns)
- Accordingly, the Level 3 cache (2MB to 256MB) refers to cache that is situated between the processor and main memory.
- Once the number of cache levels is determined, the next thing to consider is whether data (or instructions) can exist in more than one cache level.

A Real-World Example – Pentium Memory